[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 11: Stream Processing

Posted by CHARLES *on* 2020-05-01

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

A complex system that works is invariably found to have evolved from a simple system that works. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work.

—John Gall, Systemantics (1975)

- Batch process is under the assumption all the data is **Bounded**, which means we know the finite size of the data we are dealing with, so it is known when the job is finished.
- In reality, a lot of the data is unbounded. (because data keeps generated every second). This issue will force the batch process to divide data into "chunks".
  - Which means the result/derived data is **delayed** based on the interval you are chosen. (this is too slow for impatient users)
- Continuously data processing without interruptions(break into chunks) is the idea behind **streaming processing**.
  - Think of "Stream" as a never stop flow of water/river that keep feeding data in;
  - E.g. (stdin/stdout, file inputstream, TCP stream etc.)
  - "Event Stream" as a data management mechanism

Transmitting Event Streams

- Batch processing: Input are **Files** vs. Streaming Processing: Input are **Events**;
- What is **Event**: a small, self-contained, immutable object containing the details of something that happened at some point in time.

- Batch vs. Stream processing is kind like the difference between Pull and Push.
  - Batch: is the consumer keep Pull event from DB.
  - Stream: is the DB keep push Event to Consumer.
- Traditional DB/(RDMS) is not designed for “Stream/Event” processing
- **Message Systems**:
  - A producer sends a message containing the event, which is then pushed to consumers.
  - MQ vs. Unix Pipe or TCP
    - MQ allows many-to-many relationships (Producer vs. Consumer)
    - As Unix Pipe & TCP is usually one-to-one
  - What happens if the producers send messages faster than the consumers can process them? Three options
    - Drop message; Queue ; backpressure (flow control)
    - What if Queue is full ?
  - What happens if nodes crash or temporarily go offline—are any messages lost?
    - Still a trade off between C & A (Consistency and Availability)
  - **Direct messaging from producers to consumers**: (prone to loss data)
    - **UDP multicast**: used in financial industry for streams such as stock market feeds;
    - **Brokerless message library**: ZeroMQ.
    - **StatsD & Brubeck**: UDP messaging.
  - **Message brokers**: (aka. Message Queue, e.g. ActiveMQ, RabbitMQ)
    - A special type of DB that optimized for Message Streams;
    - Producer → Broker → Consumer
    - Better fault tolerance; Message could be persist to disk;
    - It all asynchronously;
  - **Message brokers compared to databases**: (JMS, AMQP)
    - In MQ Some even support 2PC (two-phase commit);
    - In MQ, data is deleted right-after message is been consumed;
    - In MQ, usually assume the Queue is short.
    - DB Secondary Indexes vs. MQ subset of topics
    - MQ has no support for queries, but notify clients when data change
    - E.g. RabbitMQ, ActiveMQ, HornetQ, Qpid, TIBCO Enterprise Message Service, IBM MQ, Azure Service Bus, and Google Cloud Pub/Sub.
  - **Multiple consumers**:
    - **Load balancing**: arbitrarily assigned to worker/consumer; Good for parallel processing expensive work-load.
    - **Fan-out**: Each message is sent to all consumers/worker; (topic subscription in JMS, exchange bindings in AMQP)

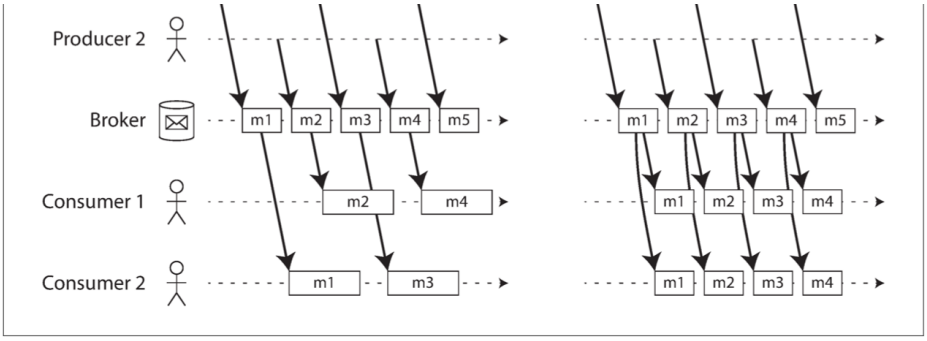

Figure 11-1. (a) Load balancing: sharing the work of consuming a topic among consumers; (b) fan-out: delivering each message to multiple consumers.

- Two patterns could be combined.
- **Acknowledgments and redelivery**:
  - **acknowledgments**: a confirmation from a client that it has finished processing a message so that the broker can remove it from the queue.
    - Note: due to network issues, the ack. Could be lost, then cause the ordering of the message change (C: In general, when you using Queue, you shouldn't have cared about the order at first place)
- **Partitioned Logs**:
  - **Transient messaging mindset**: transient operation that leaves no permanent trace. (Which is totally opposite than DB or FileSystem)
  - **MQ is NOT idempotent**: receiving a message is destructive if the acknowledgment causes it to be deleted from the broker;
  - Why can we not have a hybrid, combining the durable storage approach of databases with the low-latency notification facilities of messaging?
    - Yes, **log-based message brokers**.
  - **Using logs for message storage**:
    - Log: is simply an append-only sequence of records on disk.
    - **Log-based Message broker**: A producer sends a message by appending it to the end of the log, and a consumer receives messages by reading the log sequentially.
    - **This log can be partitioned**.
    - Each message is offset by a sequence number. (totally ordered)
      - Note: no ordering guarantee across partitions tho.

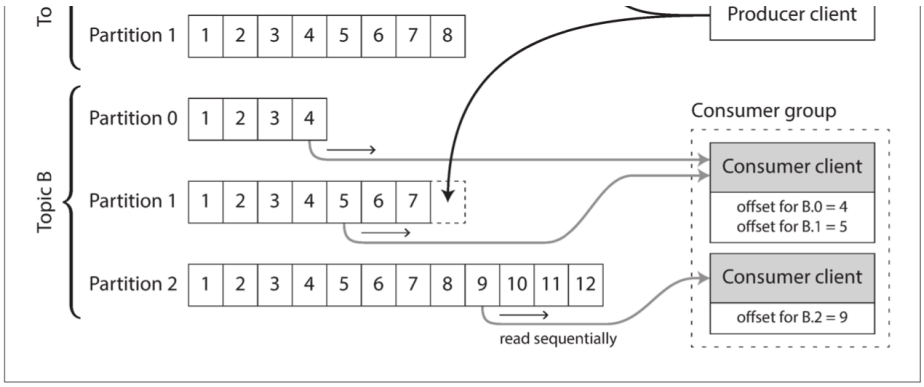

Figure 11-3. Producers send messages by appending them to a topic-partition file, and consumers read these files sequentially.

- E.g. Apache **Kafka**, Amazon Kinesis Streams, and Twitter's DistributedLog. (millions of MPS by partitioning)
- **Logs compared to traditional messaging***:
  - the broker can assign entire partitions to nodes in the consumer group, Then each client consumes all the messages in the partitions it has been assigned.
  - **JMS/AMQP style of message broker is preferable**: Message is expensive, parallel processing, order doesn't matter.
  - **Log-based approach**: high message throughput, where each message is fast to process and where message ordering is important.
- **Consumer offsets**:
  - Similar to "log sequence number" in single-leader DB replication;
  - The message broker behaves like a leader database, and the consumer like a follower.
- **Disk space usage**:
  - To prevent run out of storage, log is divided into segments, old segments are deleted or archived.
  - **"Log" is kind of like a bounded-size buffer**, if the consumer can't keep up, the old message will be discarded. (aka. Circular buffer, ring buffer)
    - E.g. 6T HDD with 150MB/s write speed can buffer up to 11 hrs of messages.
- **When consumers cannot keep up with producers**:
  - Three choices: **dropping**, **buffering** or **backpressure**(flow-control)
  - Consumers are independent from each other;
- **Replaying old messages**:
  - it is a read-only operation that does not change the log.
  - Offset is under the consumer's control. So, it has the freedom to go back to previous data/offset.
  - This made it easier for integration with other dataflows.

Databases and Streams

- DB and Stream are correlated deeper; (e.g. write is an event)

- Often need to combine several different technologies in order to satisfy their requirements. (Write/Read/Search/Analytics etc.)
- It is essential to keep all the data in-sync. if an item is updated in the database, it also needs to be updated in the cache, search indexes, and data warehouse. (usually through ETL processes.)
  - Or “Dual writes”, which prone to issue like race-conditiotion;

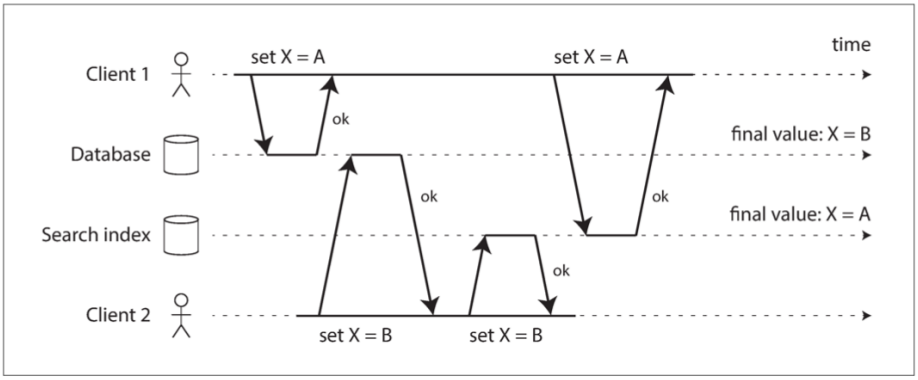

Figure 11-4. In the database, X is first set to A and then to B, while at the search index the writes arrive in the opposite order.

- The key is to determine if we can have only one “Source of Truth” (aka. Leader)
- **Change Data Capture**:
  - CDC(Change Data Capture): is a process that extracts DB changes and puts it into other systems.
    - E.g. in the form of “Stream”

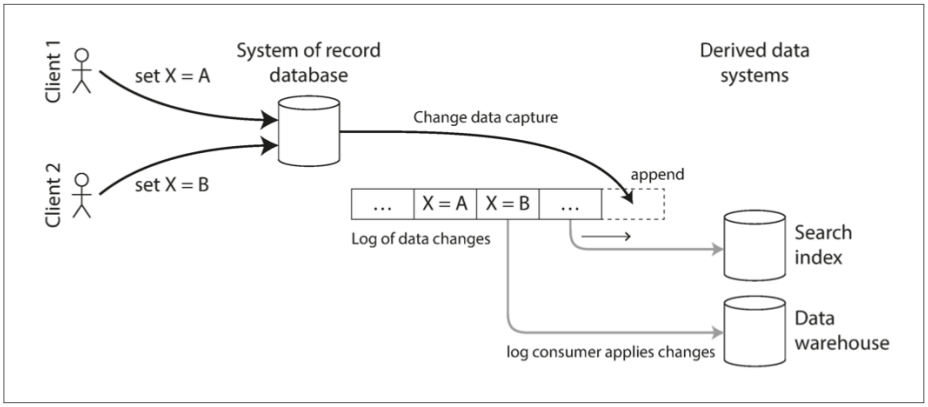

Figure 11-5. Taking data in the order it was written to one database, and applying the changes to other systems in the same order.

- **Implementing change data capture**:
  - We call any “log consumer” a “derived data system”. The idea behind CDC is to ensure all those “derived data systems” got the up-to-date changes.
  - Essentially, CDC makes one database the **leader** (the one from which the changes are captured), and turns the others into followers.

- **Initial Snapshot**:
- **Log compaction**: (e.g. Apache Kafka)
  - This is used when need add a new “derived data system”
  - This allows the message broker to be used for durable storage, not just for transient messaging.
- **API support for change streams**: (e.g. RethinkDB, FIreBase, CouchDB, Meteor, VoltDB)
  - DBs engine started to support change streams;
  - A table will hold transactions but can’t be queried.
- **Event Sourcing**:
  - **Event Sourcing**: a technique that was developed in the domain-driven design (DDD) community.
  - **CDC vs. ES**(Event Sourcing): different level of abstraction.
    - **CDC**: application isn’t aware of CDC occurring, so it happens at a lower level.
    - **ES**: reflect things that happened at the application level.
  - **ES is a powerful technique for data modeling**, because it makes more sense to record a user’s action as immutable events, rather than the effect of the actions on a mutable DB.
  - Event sourcing is similar to the chronicle data model. (or “fact table”).
  - **Deriving current state from the event log**:
    - Applications need to take the events and transform it to an application state that is suitable for the user to view. (deterministic)
  - **Commands and events**:
    - First it comes as “Command” and then after it successfully executed, it becomes “Event” which is durable and immutable.
      - when the event is generated, it becomes a fact.
    - A consumer of the event stream is not allowed to reject an event;
    - Any validation of a command needs to happen synchronously, before it becomes an event.
- **State, Streams, and Immutability**:
  - **Immutability** is also what makes event sourcing and change data capture powerful.
  - Whenever you have a state that changes, that state is the result of the events that mutated it over time.
  - **mutable state** and an **append-only log** of immutable events do not contradict each other: they are two sides of the same coin.
    - the **changelog**, represents the evolution of state over time.
  - In terms of mathematical:
    - **application state** is what you get when you **integrate** an event stream over time;
    - **a change stream** is what you get when you **differentiate** the state by time;

- *Figure 11-6. The relationship between the current application state and an event stream.*
- **Quote from Pat Helland**:
  - *Transaction logs record all the changes made to the database. High-speed appends are the only way to change the log. From this perspective, the contents of the database hold a caching of the latest record values in the logs.* ***The truth is the log****. The database is a cache of a subset of the log. That cached subset happens to be the latest value of each record and index value from the log.*
- **Log compaction**: bridging the distinction between log and DB state. it retains only the latest version of each record, and discards overwritten versions.
- **Advantages of immutable events**: (e.g. Account ledger)
  - Particularly important in financial systems, it is also beneficial for many other systems.
  - Capture more information than just the current state.
  - E.g. Shopping cart history with append-only event could help analytic in the future;
- **Deriving several views from the same event log**:
  - You can derive several different read-oriented representations from the same log of events. (C: This idea is similar to the talk about Apache Kafka,built application around the Kafka stream)
  - Having an explicit translation step from an event log to a database makes it easier to evolve your application over time. (C: enable old & new system running side by side)
  - **Command Query Responsibility Segregation (CQRS)**: you gain a lot of flexibility by separating the form in which data is written from the form it is read.
- **Concurrency control**:
  - The biggest downside of event sourcing and change data capture is asynchronous. → cause delay.
  - Potential Solutions:
    - “Reading your own writes”
    - “Implementing linearizable storage using total order broadcast”
- **Limitations of immutability**:
  - Truly deleting data could be difficult because data live in many places.
  - Deletion is more a matter of “making it harder to retrieve the data” than actually “making it impossible to retrieve the data.”

Processing Streams

- Where streams come from (user activity events, sensors, and writes to databases).
- How streams are transported (through direct messaging, via message brokers, and in event logs).
- Last question is **What can you do with Stream** ? three major options

  - 2, push the events to users directly;
  - 3, process one or more input streams to produce one or more output streams. (pipelining); **processing streams to produce other, derived streams**.
- A block of Code that processes streams so called "Operator" or "Job". (kind like Unique processes or MapReduce job)
  - Since the Stream never ends(unbounded), so sorting doesn't make sense here, neither does sort-merge joins will be used.
  - Fault-tolerance mechanisms also need to be revised.
- **Use of Stream Processing**
  - **Monitoring System**: Fraud detection, Trading System, Manufacturing System, Military and Intelligence systems.
  - **Complex event processing(CEP)**: emerged from the 90s
    - CEP allows you to specify rules to search for certain patterns of events in a stream.
  - **Stream analytics**: (e.g. Apache Storm, Spark Streaming, Flink, Concord, Samza, and Kafka Streams, Google Cloud Dataflow and Azure Stream Analytics)
    - More oriented toward aggregations and statistical metrics over a large number of events;
    - Stream analytics systems sometimes use **probabilistic algorithms**, such as Bloom filters.
  - **Maintaining materialized views**:
    - Derived data systems can be treated as maintaining materialized views.
  - **Search on streams**:
    - The percolator feature of Elasticsearch is one option for implementing this kind of stream search.
  - **Message passing and RPC**:
- **Reasoning About Time**
  - Time "window"
  - Using the timestamps in the events allows the processing to be **deterministic**.
  - **Event time versus processing time**: (e.g. Star War movies)
    - Processing may be delayed.
    - Confusing event time and processing time leads to bad data.

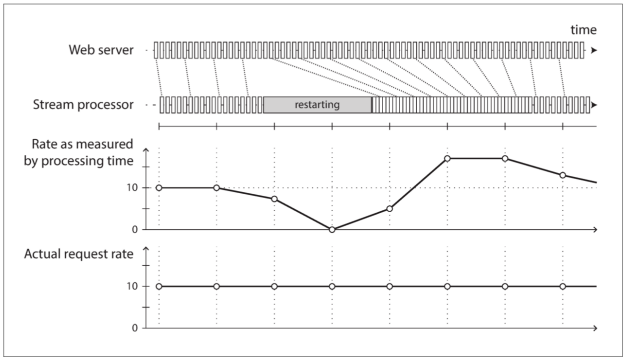

Figure 11-7. Windowing by processing time introduces artifacts due to variations in processing rate.

  - **Knowing when you're ready**:

    - 2, Publish a correction;
- **Whose clock are you using, anyway**?
    - Need address Incorrect device clocks, log three timestamps:
        - The time at which the event occurred, according to the device clock
        - The time at which the event was sent to the server, according to the device clock
        - The time at which the event was received by the server, according to the server clock
- **Types of windows**:
    - Tumbling window: fixed length, and every event belongs to exactly one window.
    - Hopping window: fixed length, but allows windows to overlap in order to provide some smoothing.
    - Sliding window: contains all the events that occur within some interval of each other.
    - Session window: has no fixed duration. But, grouping together all events relative to the same user that occur closely together in time. (e.g. website analytics)
- **Stream Joins**
    - Similar to batch jobs; However, since new events can appear anytime on a stream makes joins on streams more challenging than in batch jobs.
    - three different types of joins: **stream-stream joins**, **stream-table joins**, and **table-table joins**.
    - **Stream-stream join (window join)**:
        - a stream processor needs to maintain state.
    - **Stream-table join (stream enrichment)**:
        - Enriching the activity events with information from the database.
        - Instead of performing remote SQL queries, we can cache up a copy of DB. (In Memory hashtable or local disk index)
            - Need CDC to ensure the stream data is up-to-date;
        - A stream-table join is actually very similar to a stream-stream join, but in this case we have “table changelog stream” involved.
    - **Table-table join (materialized view maintenance)**: (e.g. Tweets)
        - it maintains a materialized view for a query that joins two tables.
    - **Time-dependence of joins**:
        - **Common**: they all require the stream processor to maintain some state based on one join input, and query that state on messages from the other join input.
        - If the state changes over time, and you join with some state, what point in time do you use for the join ? (e.g. sales Tax calculation)
        - If the ordering of events across streams is undetermined, the join becomes nondeterministic;

- **Fault Tolerance**
  - You can't wait until a stream is finished to validate its output/result, since all the stream is unbounded and will never really finish/complete.
  - **Microbatching and checkpointing**:
    - **Microbatching**: break the stream into small blocks, and treat each block like a miniature batch process. (e.g. **Spark** Streaming) usually one second interval.
      - Smaller the batches size the greater overhead.
      - Larger batches size means longer delay of results.
      - implicitly provides a tumbling window equal to the batch size
    - **Checkpointing**: triggered by barriers in the message stream, similar to the boundaries between microbatches, but without forcing a particular window size. (e.g. Apache **Flink**)
    - Both approaches won't prevent external side effects after the results have been written into External Systems.
  - **Atomic commit revisited**:
    - Achieve "Exactly-Once" processing without transactions across heterogeneous technologies.
    - **Idempotence**:
      - Distributed transactions are one way of achieving that goal, but another way is to rely on **idempotence**.
      - if an operation is not naturally idempotent, it can often be made idempotent with a bit of extra metadata. (e.g. Kafka with some offset value)
    - **Rebuilding state after a failure**:
      - keep state local to the stream processor, and replicate it periodically.
      - sometimes the state can be rebuilt from the input streams.

Summary

- Discussed **event streams**, what purposes they serve, and how to process them.
  - Similar to "batch processing" but unbounded.
  - **message brokers** and **event logs** serve as the streaming equivalent of a filesystem.
- Two types of Message brokers:
  - **AMQP/JMS-style message broker**: exact order is not important
  - **Log-based message broker**: order is kept.
    - Similar to log-structured storage engines
- Where streams come from ?
  - user activity events, sensors providing periodic readings, and data feeds (e.g., market data in finance)
  - writes to a database as a stream: capture the changelog
    - Change Data Capture
    - Event Sourcing

messages
    - searching for event patterns (complex event processing),
    - computing windowed aggregations (stream analytics),
    - keeping derived data systems up to date (materialized views).
- three types of joins:
    - Stream-stream joins
    - Stream-table joins
    - Table-table joins
- fault tolerance and exactly-once semantics
    - microbatching,
    - checkpointing,
    - transactions,
    - idempotent writes.

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST
Designing Data Intense Application – Chapter 10: Batch Processing

NEXT POST
Designing Data Intense Application – Chapter 12: The Future of Data Systems

Leave a Reply

Enter your comment here...